Experience on the Parallelization of a Cloud Modeling Code Using Computer-Aided Tools

Haoqiang Jin and Gabriele Jost*
NAS Division, NASA Ames Research Center, Moffett Field, CA 94035-1000,
{hjin,gjost}@nas.nasa.gov

Dan Johnson and Wei-Kuo Tao
Laboratory for Atmospheres, NASA Goddard Space Flight Center, Greenbelt, MD 20771
{djohnson,tao}@agnes.gsfc.nasa.gov

NAS Technical Report NAS-03-006, March 2003

Abstract

The purpose of the work reported on is two-fold: The optimization of large-scale earth science applications and the evaluation of the use of automatic parallelization tools. We have applied the CAPO computer aided parallelization tool developed at the NASA Ames Research Center to the Goddard Cumulus Ensemble (GCE) cloud modeling code. We describe how the tool was used for OpenMP parallelization and identification of performance obstacles within the code. We also give performance results for a number of different hardware platforms, discuss the results, and outline possibilities for further optimizations in the code and improvement in the tools.

1. Introduction

The Goddard Cumulus Ensemble (GCE) code developed by Tao et. al. [9] at the NASA Goddard Space Flight Center (GSFC) is used for modeling the evolution of clouds and cloud systems under large-scale thermodynamic forces. The 3-dimensionsal version of this code, GCEM3D, was chosen for this study to serve as a pilot project to evaluate the use of parallelization support tools developed at the NASA Ames Research Center (ARC) for their use on earth science modeling.

Researchers at ARC have developed parallelization tools that have successfully been applied to a number of computational fluid dynamics (CFD) codes [5]. These tools rely on in-depth source code analysis and parallelize codes with nominal user interaction. The tool-based parallelization process reduces code development time and user error. In the report we describe the experience

* Computer Sciences Corporation, NASA Contract DTTS59-99-D-00437/A618112D

of using these tools to parallelize GCEM3D and to optimize the parallelized code. We include timing comparisons for several test cases on three different machines and point out possible areas for further improvement.

In summary, we have achieved the following goals from this work:

- The OpenMP parallel code of GCEM3D was generated in a relatively short time.
- The OpenMP code ran on different types of shared-memory parallel machines, such as SGI Origin, SUN E10K, and Dell PC.
- We achieved a 12x speedup on 16 CPUs of an SGI O3K for a 130×66×34 test case and scaled up to 64 CPUs for larger test cases.
- We successfully ran a large test case of 1026×1026×34 which uses more than 7GB of memory and has not been tried before.

2. Code Description

The GCEM3D code is written Fortran 77. It contains about 18000 lines and 100 subroutines. The code contains a limited number of SGI '`DOACROSS`' directives for achieving additional performance on parallel machines. With the help of the auto-parallelization option "`-Mconcur`" from the Portland Group compiler, the performance increases by a factor of 1.7 on 2 CPUs of a PC, but the scalability is very limited beyond 4 CPUs. Besides, the format of the parallelization directives is not portable. We will refer to this version as *original* in this report.

3. Parallelization Process

The task of parallelizing the GCEM3D code was performed with the CAPO [3], computer-aided tool, which was developed at NASA Ames Research Center. CAPO utilizes the full strength of the data dependence analysis engine from CAPTools [4], developed at the University of Greenwich, and automatically inserts OpenMP [7] directives in Fortran programs. A comprehensive graphical user interface built into CAPO allows a user to guide the parallelization process to achieve high performance of the resulting code. The tool is aimed at automating the straightforward but tedious and error prone steps of the parallelization process, allowing the user to focus on the optimization of critical parts of the code. CAPO has successfully been used to parallelize a number of large-scale computational fluid dynamics (CFD) codes [5]. More information on the tool can be found in [3].

In preparation for the data dependence analysis, we noticed that a few parameters controlling the data flow in the program were hard-coded with preset values. We added dummy READ statements for these parameters in order to avoid dead-code elimination performed by the dependence analysis engine. This helps to minimize changes to the original source code. The dependence analysis took 6.2 hours of CPU time on a Sun UltraSparc 450MHz workstation.

In order to insert directives automatically into the code, CAPO must examine the loops for potential data dependences that might prohibit parallelization. The tool, just like any compiler,

has to be conservative in its assumption about data dependences. To avoid the generation of incorrect code, data dependences have to be assumed unless proven otherwise. A very complex code structure may inhibit tests for independence, or input parameters may be involved that are only known at runtime. Unlike a compiler, CAPO allows the user to provide knowledge about code structures or input parameters in order to remove unnecessary data dependences. It is the task of the user to improve the data dependence results by pruning *false* data dependences, which in turn will improve the level of parallelization. CAPO aids the user in this task by indicating the obstacles for each loop that has not been parallelized. The user can then decide if and how to remove these obstacles. In case of GCEM3D we used CAPO's directive browser (see Figure 1 for a snapshot) to remove many data dependences that were assumed mainly because of implicit `EQUIVALENCE` conditions existed among many common blocks. These common blocks are used mostly as storage space for working arrays which are declared differently in different routines. We eliminated these unnecessary dependences among these working arrays.

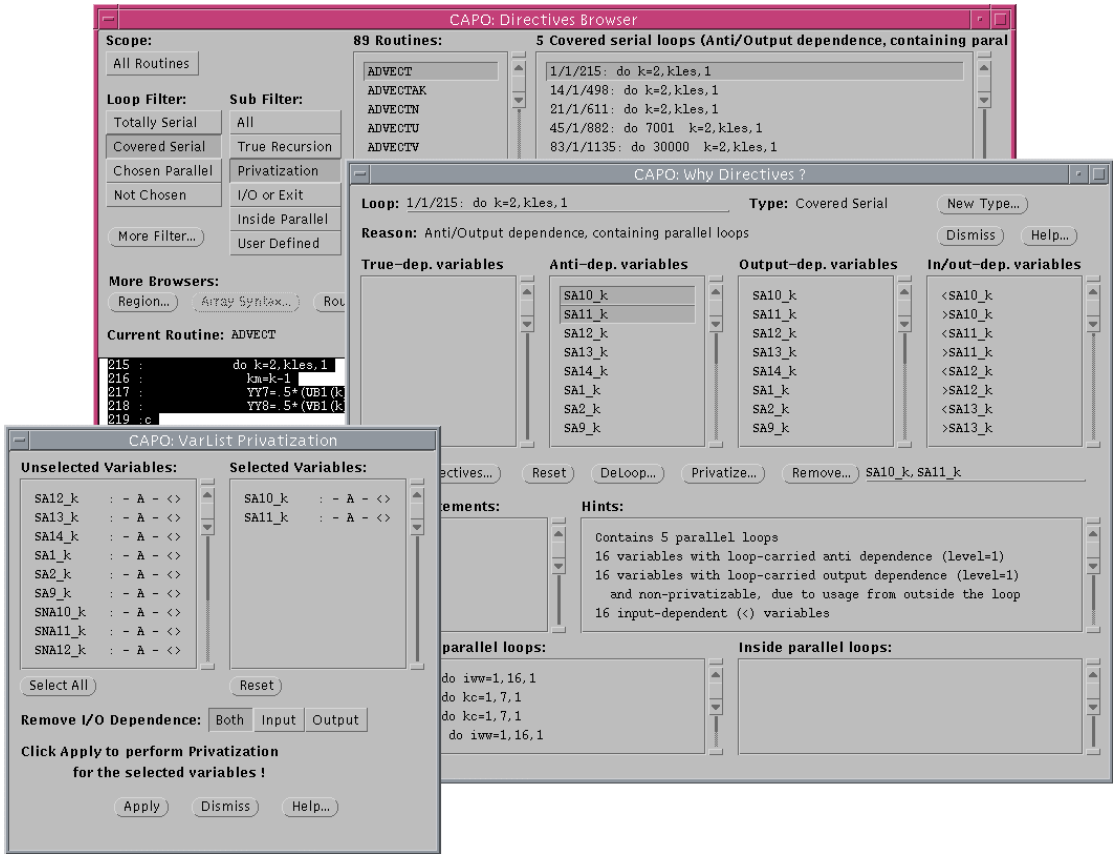

Figure 1: Snapshot of the CAPO graphical user interfaces for a parallelization session. The main browser, Directives Browser, lists loops and their types in a program; the Why Directives browser presents some detailed information on why a given loop is not parallel and indicates the problematic variables. The user can inspect this information and enforce a particular type (such as private) for the selected variables so that obstacles in parallelization of the loop could be removed.

The parallelization strategy used by CAPO targets outer-most loops, which, if successful, yields coarse parallel granularity and little OpenMP overhead. In the case of GCEM3D the outer-most loop is, in many cases, the K loop. In the cases where the K loop has data dependences, the next level, usually the J loop, is chosen for parallelization. During the final stage of the OpenMP code generation by CAPO, we verified that the proper loops were selected for parallelization using the directives browser. The generated directives were then transferred back to the original code in order to preserve the structure of the original code.

4. Parallel Code Optimization

The initial OpenMP parallel code generated by CAPO has limited performance. For example, the improvement was only a factor of 4.2 on 16 CPUs. This is mainly due to certain structures in the original code that prevented CAPO from performing further parallelization or selecting a better parallelization strategy. Most of the transformations to the code structure were done by hand, however, CAPO has been used to identify where code restructuring is required in order to improve the performance. In the following subsections we will describe some of the key code sections where additional optimizations were performed.

4.1. FFT

GCEM3D uses an FFT algorithm (routine SLVPI) that first applies to the X (or I) dimension and then to the Y (or J) dimension. An efficient parallel implementation of FFT on multiple dimensions usually contains the following two characteristics:

- proper blocking for cache size, and
- 1-D FFT inside a parallel loop.

The original code has none of the above characteristics. We adopted a method that was implemented in the FT kernel of the NAS Parallel Benchmark ([1],[2]). The new code has the following structure:

```
DO K=2,KLES
   copy a slice of (I,J) block to working arrays AR and AI
   CALL FFTX(AR,AI)
   copy results back to the original array
END DO
```

The size of the working block was chosen so that the data could fit into a typical cache size. The outer K loop was selected for parallelization with the working arrays AR and AI as local variables. The routine FFTX performs 1-D FFT.

4.2 The Radiation Code

The radiation effect is calculated in routine PRADRAT which calls routine RADRAT. There is one call to RADRAT for loop index J = 2 and then a second call to RADRAT within a loop where loop index J ranges from 2 to JLES. The corresponding call graph is displayed in Figure 2. It is most efficient to parallelize the loop outside of the call to RADRAT. The CAPO directives browser indicates two major obstacles to the parallelization of the critical loop:

- A list of array variables carrying true dependences, and
- I/O within the loop (by a call to routine FITO3).

Inspection of the arrays shows that they are 2-dimensional arrays. They are not dependent on the parallelized dimension in J direction. For each J, they hold a 2-dimensional slice of a 3-dimensional array. The CAPO user interface allows privatizing all of these variables and removing the dependences. The common blocks containing the 2-dimensional arrays can be declared as THREADPRIVATE in routine PRADRAT.

Inspection of the I/O shows that this occurs only once, to initialize certain arrays. Parallelization of the critical loop with I/O can be forced by using the CAPO user interface, or the call to FITO3 can be moved outside of the parallel loop. We decided to use the second approach (see Figure 2).

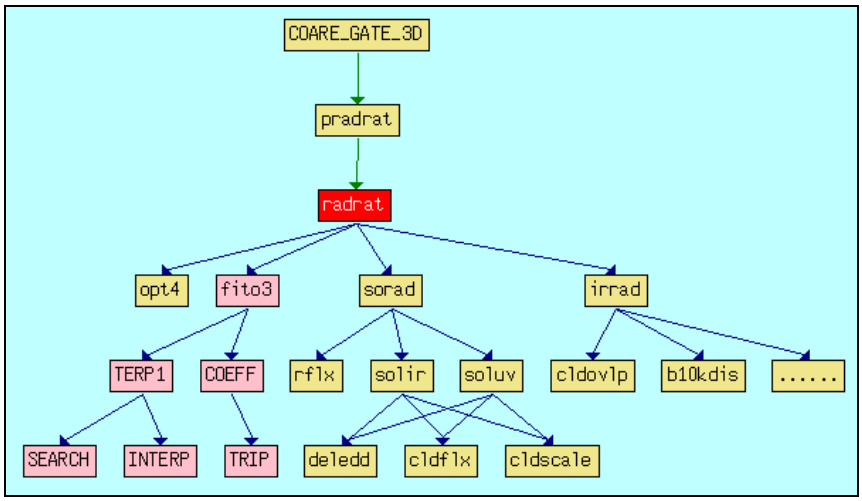

Figure 2: The call path followed by subroutine RADRAT in the original code. The branch led by FITO3, which contains I/O is only called once and was moved outside of the J loop in PRADRAT so that a more efficient parallelization of the J loop can be obtained.

We removed the SAVE statements from routines PRADRAT, RADRAT, and OPT4 manually, since a global SAVE will make all variables within a subroutine to be shared and in general prevents parallelization.

Compared to the initial CAPO generated version where many inner loops below RADRAT were chosen for parallelization, the coarse-grained parallelization in the new version improves the scalability of PRADRAT from 4.6 to 16.4 on 16 CPUs. This super-linear speedup reflects very small overhead from parallelization and the increased overall cache size as the number of CPUs increases.

The effect of outer versus inner loop level parallelization is quantified in Figure 3. The figure shows the average time distribution across various thread states for routine PRADRAT running with 8 threads. We categorize the states of a thread as *running, idle, synchronizing,* and *fork/join*. When outer level parallelization is performed, we have only one parallel region. After a certain start-up time all threads spend their time running, i.e. they do useful work. For the case of inner loop level parallelization, there are many more parallel regions involved. The threads have to be forked and joined at the beginning and end of these parallel regions, which results in considerable synchronization overhead.

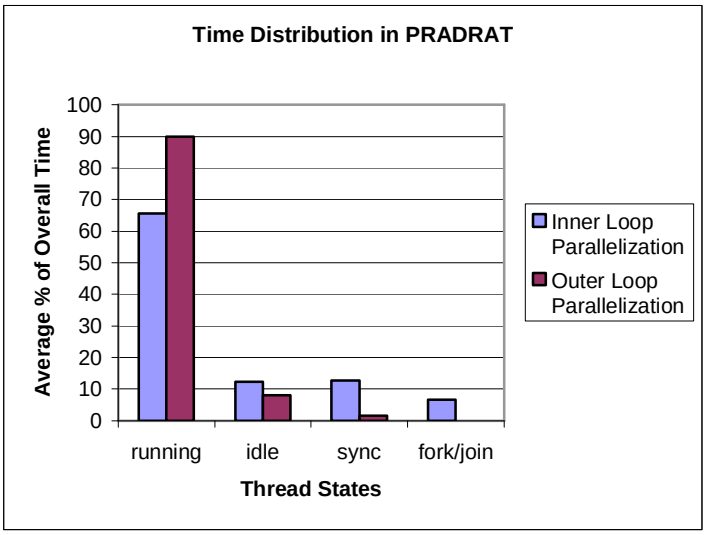

Figure 3: Comparison of outer versus inner loop level parallelization for an 8-thread run.

A similar situation occurred in a less time-consuming routine PBLIN in which the routine SFFLUX is called in a J loop. The initial version serialized this loop. In order to parallelize this loop, the global SAVE statements in the subsequent calls along the call branch (see Figure 4) was manually removed. CAPO was able to place the THREADPRIVATE directive for the common blocks shared among the subroutines in the branch.

4.2. Advection Calculation

In the original version a commonly used code structure looks like:

```
DO J=2,JLES
  TM(I,J)=..AK(I,J,1)
ENDDO
DO K=2,KLES
  DO J=2,JLES
    TP(I,J)=..AK(I,J,K)
    U(I,J,K)=..(TM(I,J)
                - TP(I,J))..
    TM(I,J)=TP(I,J)
  ENDDO
ENDDO
```

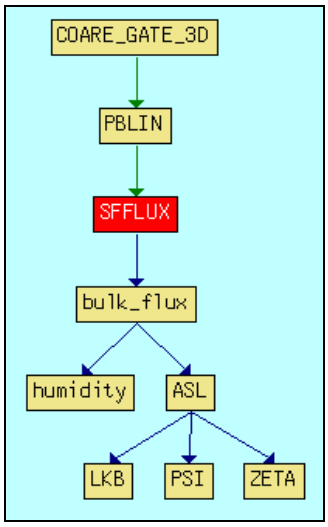

Figure 4: The call path followed by the call to SFFLUX in subroutine PBLIN.

This code works fine sequentially since the memory is efficiently reused. Due to the data usage dependence carried by TM in the K loop, the J loop is chosen for parallelization. There are two issues with this method: 1) more overhead is associated with the OpenMP PARALLEL DO because the parallelization is inside the K loop, and 2) there is potential cache trashing due to the update of smaller shared arrays TP and TM and more memory traffic. The solution is to swap the K and J loops and pre-calculate a local variable like the following:

```
DO J=2,JLES
  DO K=1,KLES
    Y1K(I,K)=..AK(I,J,K)
  ENDDO
  DO K=2,KLES
    KM=K-1
    U(I,J,K)=..(Y1K(I,KM) - Y1K(I,K))..
  ENDDO
ENDDO
```

The outside J loop is then parallelized and the use of Y1K as a local array improves cache utilization and memory traffic. The timing profile has indicated the solution improves the speedup for ADVECTU(V,W) from 5 to 10 on 16 CPUs.

The other optimization involves replacing an array element update with a scalar update in routine ADVECT so that when the outer K loop is chosen for parallelization the cache

invalidation caused by updating the shared variable can be reduced. An example is shown in the following code section:

Original	New

`DO K=2,KLES`
`  DO J=2,LES`
`  DO I=2,ILES`
`  SA1(K) = SA1(K) + ..`
`  ENDDO`
`  ENDDO`
`ENDDO`
	`DO K=2,KLES`
	`  SA1_K = SA1(K)`
	`  DO J=2,LES`
	`  DO I=2,ILES`
	`  SA1_K = SA1_K + ..`
	`  ENDDO`
	`  ENDDO`
	`  SA1(K) = SA1_K`
	`ENDDO`

5. Discussion of Timing Results

We tested the CAPO generated OpenMP version of GCEM3D for four test cases on four types of machines and compared with the original version. Detailed results and the test environment are given in Appendix. We will summarize the main results in this section.

The improvement of the new parallel code generated with CAPO over the original version is illustrated in Figure 5 for a 130×66×34 case on the SGI Origin 3000. The original version does not scale beyond 4 CPUs and the best speedup is only 1.5 on 8 CPUs. The CAPO version performed similarly sequentially, but achieved a speedup of 12.4 on 16 CPUs. This is a factor of 8.2 improved over the best timing from the original version.

As mentioned in Section 2, the original version contains a limited number of parallel directives relying on automatic parallelization performed by the compiler. Such an "auto" parallelizing option is available in the vendor compilers (SGI, Sun and PGI)

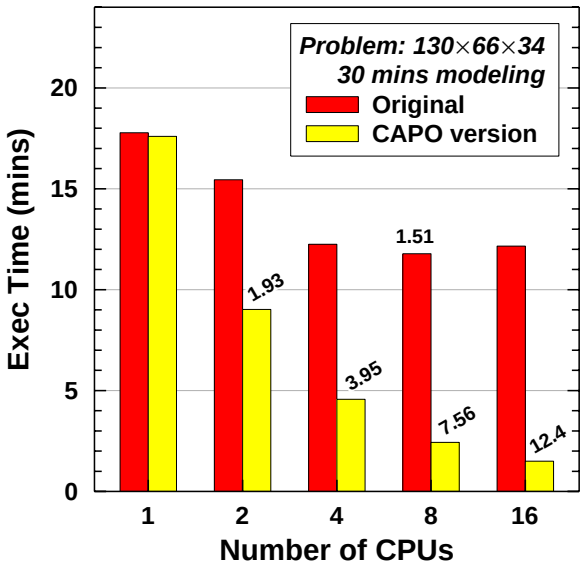

Figure 5: Comparison of the new parallel code (lighter bar) with the original version (darker bar) for a (130×66×34) test case. The speedup relative to the single CPU timing is indicated in the graph. The timings were obtained on an SGI O3K.

we tested. Automatic parallelization performed by Sun and PGI compilers does improve the performance (see Tables 4 and 5 in Appendix) and the best result is a speedup of 2.74 on 8 CPUs. The SGI automatic parallelization, however, brought little improvement. In some cases performance degraded. In Figure 5, we included the results from the original version compiled without the "auto-parallelizing" option.

The fact that the original code contained large sections of sequential code, which the compiler could not properly exploit, is illustrated by Figures 6 and 7. These figures were obtained by running the Paraver [8] performance analysis tool. Figure 6 shows the timeline of the first few iterations of a loop in routine PRADRAT for 8 threads. Dark shading indicates a thread running useful user code. Light shading indicates that the thread performs "non-useful" operations such as idling, synchronization, or fork/join operations.

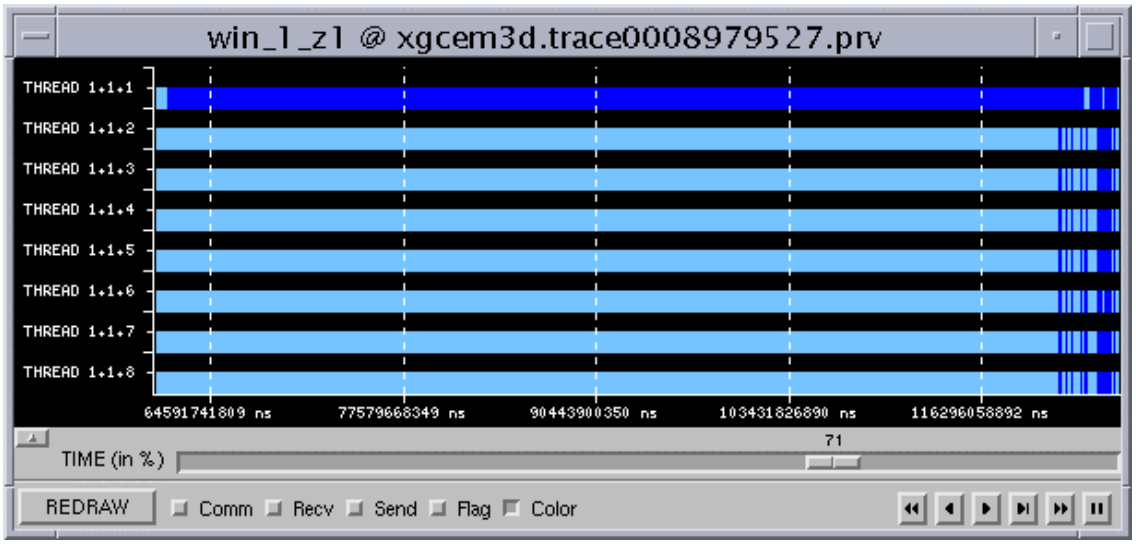

Figure 6: Thread timeline for the original GCEM3D running on 8 CPUs. Dark shading indicates useful time for a thread. Light shading indicates non-useful time. Due to the fact that large sections of the code are not parallelized, only one of the master thread performs useful operations while the others are idling or synchronizing.

Figure 6 shows that for a large amount of time only one thread performs useful operations, while the remaining threads are spending their time in synchronization. This is due to the fact that routine PRADRAT was not parallelized in the original code. Only the master thread is active for this routine. In the CAPO-parallelized version this problem is overcome, as can be seen in Figure 7 where the dark shading indicates that now all threads are running user code.

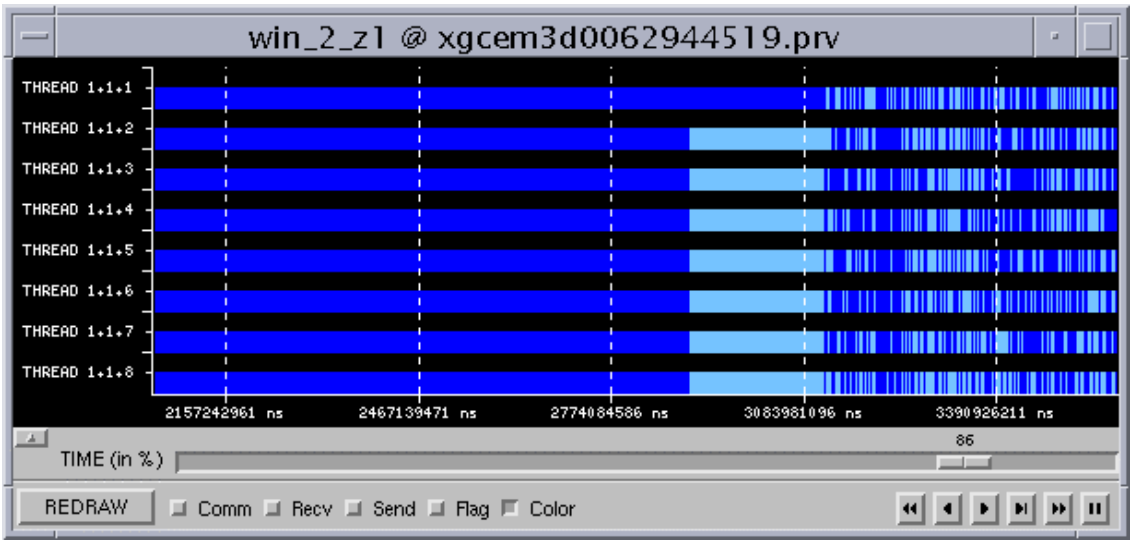

Figure 7: Thread timeline for the CAPO parallelized GCEM3D running on 8 CPUs. Similar to Figure 6, dark shading indicates that a thread is running user code. The amount of useful time for each thread is greatly increased after improved parallelization.

6. Future Improvements

The current OpenMP version of GCEM3D has limited scalability beyond 64 CPUs, partly due to the outer-loop parallelization strategy used in the code. For the large case of 1026×1026×34, it took 62 minutes to model a 10-minute result on 32 CPUs. On 64 CPUs, the time only went down to 47 minutes. This is due the fact that the outer loop is only 34 iterations long. Clearly there is enough work for more CPUs, but a different parallelization strategy is required in order to have an efficient parallelization. One possibility is to use OpenMP directives on the J dimension for larger cases. Ideally the code could be restructured such that the K dimension becomes the first dimension and the innermost loop. This will most likely yield high performance and good OpenMP scalability, however, it will require a lot of rewriting of the code, since tools are still very limited in the areas of code restructuring. Support of automatic code restructuring is one of the areas to be improved in the tools.

Other possible future work to improve the performance would be to perform domain-decomposition (on the J dimension, for example) to create a message passing version based on MPI [6] of the application. The MPI version has the potential for scalability not only on shared-memory machines, but also on clusters of SMP nodes. Generating a message-passing version will involve more manpower. It is possible to apply CAPTools to generate a message-passing parallel code, but this requires a certain level of expertise in producing such a code.

Acknowledgements

This work was supported in part by NASA contracts NAS 2-14303 and DTTS59-99-D-00437/A61812D with Computer Sciences Corporation/AMTI.

References

[1] D. Bailey, J. Barton, T. Lasinski, and H. Simon (Eds.), "The NAS Parallel Benchmarks," *NAS Technical Report RNR-91-002*, NASA Ames Research Center, Moffett Field, CA, 1991.

[2] D. Bailey, T. Harris, W. Saphir, R. Van der Wijngaart, A. Woo, and M. Yarrow, "The NAS Parallel Benchmarks 2.0," *NAS Technical Report NAS-95-020*, NASA Ames Research Center, Moffett Field, CA, 1995. http://www.nas.nasa.gov/Software/NPB.

[3] CAPO, http://www.nas.nasa.gov/Tools/CAPO.

[4] C.S. Ierotheou, S.P. Johnson, M. Cross, and P. Leggett, "Computer Aided Parallelisation Tools (CAPTools) – Conceptual Overview and Performance on the Parallelisation of Structured Mesh Codes," *Parallel Computing*, 22 (1996) 163-195. http://captools.gre.ac.uk/

[5] H. Jin, M. Frumkin and J. Yan. "Automatic Generation of OpenMP Directives and Its Application to Computational Fluid Dynamics Codes," in *Proceedings of Third International Symposium on High Performance Computing* (ISHPC2000), Tokyo, Japan, October 16-18, 2000.

[6] MPI 1.1 Standard, http://www-unix.mcs.anl.gov/mpi/mpich.

[7] OpenMP Fortran Application Program Interface, http://www.openmp.org/

[8] Paraver, http:/www.upc.cepba.es/paraver

[9] W.-K. Tao, "Goddard Cumulus Ensemble (GCE) Model: Application for Understanding Precipitation Processes, AMS Meteorological Monographs," *Symposium on Cloud Systems, Hurricanes and TRMM*, 2002 (in press).

Appendix

This appendix summarizes the timing results of the GCEM3D OpenMP version created from the CAPO parallelization tool. It includes results for four test cases on five different shared memory machines.

A.1 Test conditions

The test cases and machines are summarized in Tables 1 and 2, respectively.

Table 1: Parameters used in the four test cases. The memory usage is estimated from arrays statically allocated in common blocks, assuming `REAL*4` for the floating point numbers. The smallest case (0) is mostly used in the code development. DT is the time step used in the tests.

	NX	NY	NZ	DT (sec)	Memory
Case 0	130	66	34	12	77MB
Case 1	258	258	34	12	440MB
Case 2	514	514	34	12	1.8GB
Case 3	1026	1026	34	12	7.1GB

Table 2: Five types of shared memory parallel machines used in the current work.

Name	Type	CPU, MHz	L2	#Nodes	#CPUs per node	Memory per node
lomax	SGI O2K	R12K, 400	8MB	256	2	768MB
crick	SGI O3K	R12K, 400	8MB	128	4	2GB
simak	Sun E10K	UltraSparc, 333	1MB	1	16	2GB
zan	Dell PC	PIII, 933	256KB	1	2	1GB
ibm02*	IBM p690	Pwr4, 1300	1.5MB	1	32	32GB

*The IBM machine contains a 128MB L3 cache (memory buffer) for each multi-chip module (MCM) shared among 8 CPUs.

A.2 Compilation and run-time flags

In all cases, the "`OMP_NUM_THREADS`" environment variable was used to control the number of threads (or CPUs) for running a program.

On SGI Origin 2000 and 3000 running IRIX 6.5:

Compiler: SGI MIPSpro f77 compiler, version 7.3.1.2m

Compilation option: "`-64 -O3 -mp`" for compiling OpenMP or SGI `DOACROSS` programs.

Additional option "`-apo`" was also used for the MIPSpro compiler to automatically parallelize the original version in conjunction with the directives already being inserted by hand.

Run-time environment: mpt-1.4.0.2

To avoid the slave stacksize overflow due to the increased local variable sizes in Case 3, the default stack size for the slave threads was increased by

setenv MP_SLAVE_STACKSIZE 100000000 (to 100MB)

On Sun Enterprise 10000 running Solaris 7:

Compiler: Sun Workshop HPC f95 compiler, version 6.1

Compilation option: "`-fast -openmp`" for compiling the OpenMP program. Additional option "`-autopar`" was also used for the Sun compiler to perform automatic loop parallelization for the original code in conjunction with those directives inserted by hand (The `DOACROSS` directives were replaced by `OMP PARALLEL DO`).

Run-time environment: Sun Workshop for HPC

To avoid the slave stacksize overflow due to the increased local variable sizes in the large case, the default stack size for the slave threads was increased by

setenv STACKSIZE 16384 (to 16MB)

On Dell PC running RedHat Linux 7:

Compiler: Portland Group PGI f77, version 3.2-4

Compilation option: "`-fast -mp`" for compiling the OpenMP program. Additional option "`-Mconcur`" was also used for the original code to allow the compiler to automatically exploit additional parallelism other than those directives inserted by hand.

On IBM p690 running AIX 5:

Compiler: xlf_r, version 7.1

Compilation option: "`-qnosave –O3 –qsmp=omp`" for compiling the OpenMP program. An auto-parallelization option is also available, but was not used.

To avoid the slave stacksize overflow due to the increased local variable sizes in the large case, the default stack size for the slave threads was increased by

setenv XLSMPOPTS stack=16000000 (to 16MB)

A.3 Timing results

The timings are wall-clock time measured from the beginning to the end of program executions, which include the time spent in the file input/output. The results are summarized in Tables 3-5 for Case 0, Table 6 for Case 1, and Table 7 for Cases 2 and 3. The timings are reported in minutes except for the finer time steps in Table 5.

Table 3: Comparison of the original and OpenMP versions for Case 0, 30-minute simulation on crick (SGI O3K). The original version was tested under two conditions: compiled with "`-mp`" and compiled with "`-mp -apo`". See A.2 for the description of these compilation flags. The speedup is calculated relative to the single CPU timing.

	Original version				OpenMP version	
#CPUs	Time (min)	Speedup	Time -apo	Speedup	Time (min)	Speedup
1	17.8	1.00	18.1	1.00	17.4	1.00
2	15.5	1.15	13.9	1.30	9.0	1.93
4	12.3	1.45	12.3	1.47	4.4	3.95
8	11.8	1.51	17.1	1.06	2.3	7.56
16	12.2	1.46	22.4	0.81	1.4	12.4
32					1.2	14.1

Table 4: Comparison of the original and OpenMP versions for Case 0, 10-minute simulation on simak (Sun E10K). The original version was tested under two conditions: compiled with "`-openmp`" and compiled with "`-openmp -autopar`". See A.2 for the description of these compilation flags. The speedup is calculated relative to the single CPU timing.

	Original version				OpenMP version	
#CPUs	Time (min)	Speedup	T-autopar	Speedup	Time (min)	Speedup
1	20.1	1.00	19.5	1.00	19.3	1.00
2	14.3	1.41	12.5	1.55	9.73	1.98
4	11.7	1.72	8.54	2.28	4.98	3.87
8	10.4	1.93	7.12	2.74	2.58	7.47
12	10.1	1.99	7.84	2.48	1.99	9.70

Table 5: Detailed break-down of timings in seconds spent in typical time steps (TS) for Case 0, 30-minute simulation on zan (Dell PC). The original version was tested under two conditions: compiled with "`-mp`" and compiled with "`-mp -Mconcur`". See A.2 for the description of these compilation flags. The speedup is given in the last row.

	Original version			OpenMP version	
#CPUs	1	2	2-Mconcur	1	2
1st TS	184.6	185.0	120.3	183.8	95.8
15th TS	15.7	12.3	12.3	15.5	8.83
50th TS	94.2	93.2	60.7	93.8	48.4
per TS	6.45	4.26	3.98	6.26	3.75
Total	25.1 min	19.4 min	16.1 min	24.5 min	14.0 min
Speedup	1.00	1.29	1.56	1.00	1.75

Table 6: Case 1: 10-minute simulation on lomax (SGI O2K) and simak (Sun E10K), and 30-minute simulation on lomax. The speedup of the OpenMP code is calculated from the single CPU timing.

	lomax, 10 mins		simak, 10 mins		lomax, 30 mins	
#CPUs	Time (min)	Speedup	Time (min)	Speedup	Time (min)	Speedup
1	68.93	1.00	177.1	1.00	153.2	1.00
2	39.78	1.73	89.3	1.98	81.6	1.88
4	20.00	3.45	45.6	3.89	41.6	3.68
8	10.43	6.61	23.4	7.56	21.1	7.27
12	7.48	9.22	16.6	10.7	15.4	9.95
16	5.83	11.8			11.3	13.6
32	3.84	18.0			6.91	22.2
64	3.60	19.2			6.28	24.4

Table 7: Cases 2 and 3: 10-minute simulation on crick (SGI O3K). The speedup is calculated relative to the 1 CPU timing for Case 2, assuming an ideal speedup of 4 on 4 CPUs for Case 3.

	crick, Case 2		crick, Case 3	
#CPUs	Time (min)	Speedup	Time (min)	Speedup
1	293.0	1.00		
2	153.8	1.91		
4	78.98	3.71	371.3	4.00
8	42.45	6.90	195.8	7.59
12	28.75	10.2	140.3	10.6
16	23.58	12.4	100.8	14.7
32	14.27	20.6	62.05	23.9
64	12.02	24.4	46.87	31.7

Table 8: Simulations performed on the IBM machine, ibm02: 30 minutes in Case 1 and 10 minutes in Case 2. The speedup is calculated relative to the 1 CPU timings.

	ibm02, Case 1		ibm02, Case 2	
#CPUs	Time (min)	Speedup	Time (min)	Speedup
1	69.04	1.00	126.57	1.00
2	34.66	1.99	63.39	2.00
4	17.45	3.96	32.19	3.93
8	8.98	7.69	16.50	7.67
12	6.33	10.91	11.51	11.00
16	4.92	14.04	8.79	14.40
32	3.07	22.49	5.15	24.58